

Question 1. (4 points) Consider the following Python code.

```
for j in range(n):
```

```
    i = 1
```

```
    while i < n:
```

```
        print(i, j)
```

```
        i = i * 2
```

```
        i = i * 2
```

$$O(n \log_2 n)$$

What is the big-oh notation $O()$ for this code segment in terms of n ?

Question 2. (4 points) Consider the following Python code.

```
for i in range(n):
```

```
    k = n
```

```
    while k > 1:
```

```
        k = k // 2
```

```
        print(k)
```

```
    for j in range(n):
```

```
        print(i, j)
```

$$O(n^2)$$

What is the big-oh notation $O()$ for this code segment in terms of n ?

Question 3. (4 points) Consider the following Python code.

```
def main(n):
```

```
    for i in range(n):
```

```
        doSomething(n)
```

```
def doSomething(n):
```

```
    for j in range(n*n):
```

```
        doMore(n)
```

```
def doMore(n):
```

```
    for k in range(n*n):
```

```
        print(k)
```

```
main(n)
```

$$O(n^5)$$

What is the big-oh notation $O()$ for this code segment in terms of n ?

Question 4. (5 points) Suppose a $O(n^4)$ algorithm takes 10 second when $n = 1000$. How long would the algorithm run when $n = 10,000$?

$$O(n^4) \Rightarrow T(n) = c n^4$$

$$T(1000) = c 1000^4 = 10 \text{ sec}$$

$$c = \frac{10 \text{ sec}}{1000^4} = \frac{10 \text{ sec}}{10^{12}} = \frac{1 \text{ sec}}{10^{11}}$$

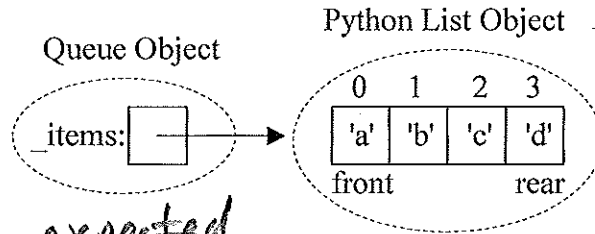
$$\begin{aligned} T(10000) &= c 10000^4 = c 10^{16} \\ &= \left(\frac{1 \text{ sec}}{10^{11}} \right) 10^{16} \\ &= 10^5 \text{ sec} = 100,000 \text{ sec} \end{aligned}$$

Question 5. (8 points) Why should a method/function having a "precondition" raise an exception if the precondition is violated?

To help the programmer immediately know that an error occurs.

Question 6. A FIFO queue allows adding a new item at the rear using an enqueue operation, and removing an item from the front using a dequeue operation. One possible implementation of a queue would be to use a built-in Python list to store the queue items such that

- the front item is always stored at index 0,
- the rear item is always at index $\text{len}(\text{self_items}) - 1$ or -1



a) (6 points) Complete the big-oh $O()$, for each Queue operation, assuming the above implementation. Let n be the number of items in the queue.

isEmpty	enqueue(item)	dequeue	peek - returns front item without removing it	__str__	size
$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$

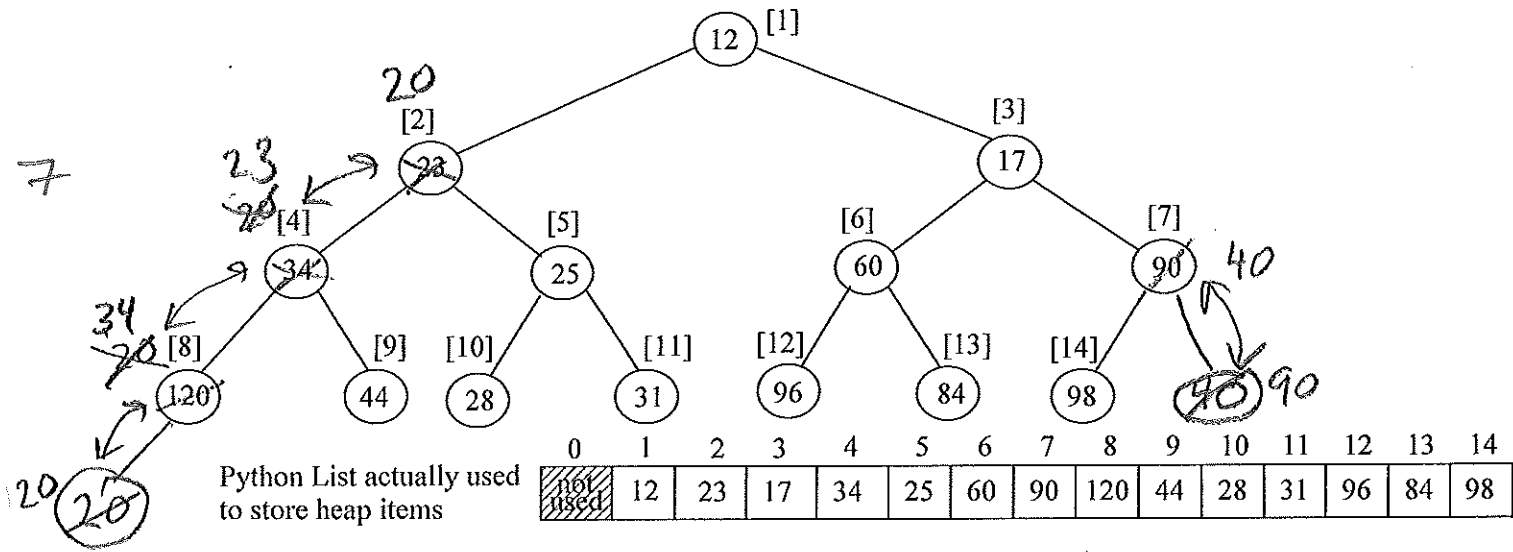
b) (8 points) Complete the method for the dequeue operation, including the precondition check to raise an exception if it is violated.

```
def dequeue(self):
    """Removes and returns the Front item of the Queue
    Precondition: the Queue is not empty.
    Postcondition: Front item is removed from the Queue and returned"""
    if len(self._items) == 0:
        raise Exception("cannot dequeue from empty queue")
    return self._items.pop(0)
```

c) (8 points) Complete the method for the __str__ operation,

```
def __str__(self):
    """ Returns a string representation of items from front to rear. """
    strResult = "(head) " (front)
    for item in self._items:
        strResult += str(item) + " "
    return strResult += "(rear) "
```

Question 7. Consider the binary heap approach to implement a priority queue. A Python list is used to store a *complete binary tree* (a full tree with any additional leaves as far left as possible) with the items being arranged by *heap-order property*, i.e., each node is \leq either of its children. An example of a *min heap* "viewed" as a complete binary tree would be:

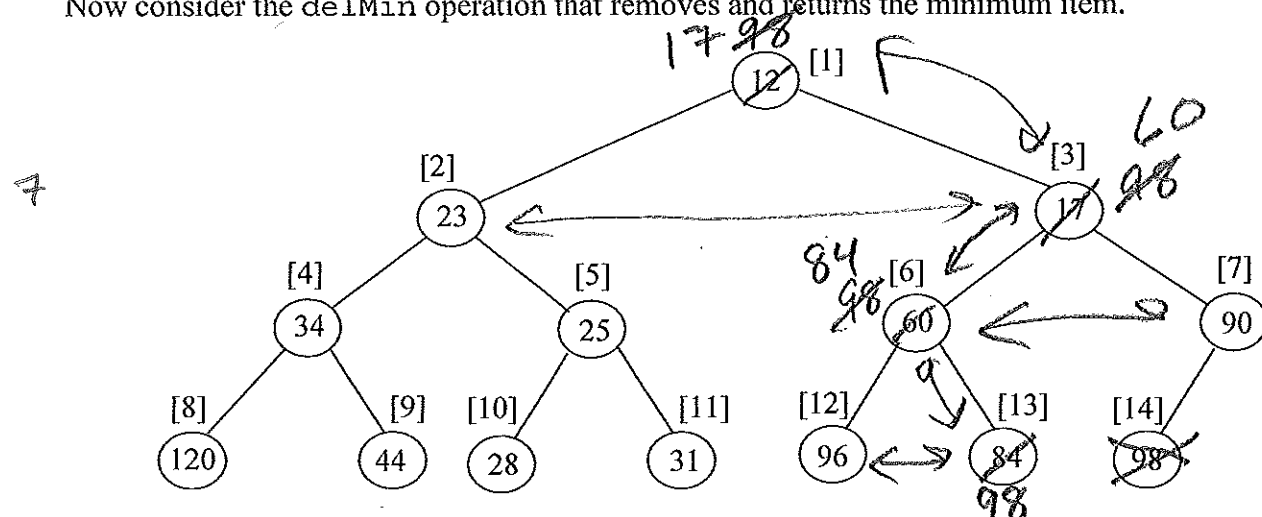


a) (3 points) For the above heap, the list indexes are indicated in []'s. For a node at index i , what is the index of:

- its left child if it exists: $i * 2$
- its right child if it exists: $i * 2 + 1$
- its parent if it exists: $i // 2$

b) (7 points) What would the above heap look like after inserting 40 and then 20 (show the changes on above tree)

Now consider the `delMin` operation that removes and returns the minimum item.



c) (2 point) What item would `delMin` remove and return from the above heap? 12

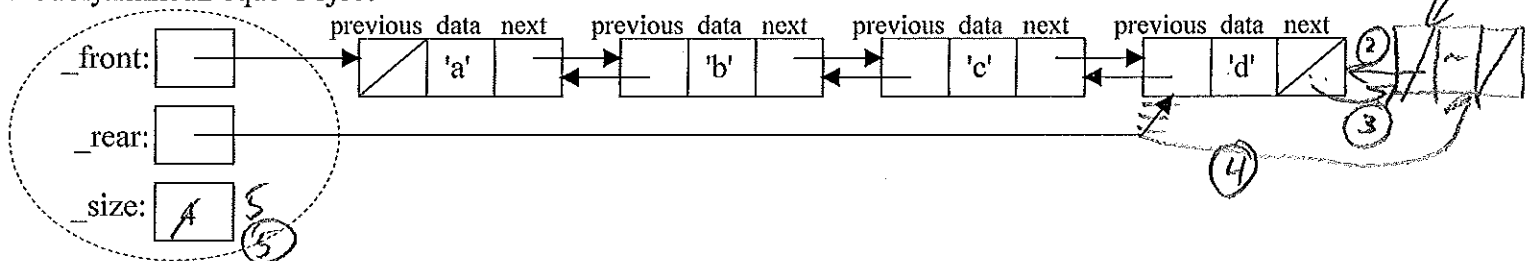
d) (7 points) What would the heap look like after `delMin`? (show the changes on tree in the middle of the page)

e) (6 points) Performing 20,000 `inserts` into an initially empty binary heap takes 0.23 seconds. Now, if we perform 20,000 `delMin` operations, it takes 0.39 seconds. Explain why 20,000 `delMin` operations take more time than the 20,000 `insert` operations?

An inserted item only needs a single comparison with its parent to move up a level, and it typically only move partially up tree. The `delMin` moves a leaf up to the root so it is likely to percolate down to being a leaf, and at each level 2 comparisons (two child then compare to min. child) are needed per level.

Question 8. The `Node2Way` and `Node` classes can be used to dynamically create storage for each new item added to a Deque using a doubly-linked implementation as in:

DoublyLinkedListDeque Object



a) (6 points) Complete the big-oh expected $O()$, for each `DoublyLinkedListDeque` operation, assuming the above implementation. Let n be the number of items in the `DoublyLinkedListDeque`.

<code>isEmpty</code>	<code>addRear</code>	<code>removeRear</code>	<code>addFront</code>	<code>removeFront</code>	<code>__str__</code>
$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$

b) (16 points) Complete the `addRear` method for the above `DoublyLinkedListDeque` implementation.

```
class DoublyLinkedListDeque(object):
    """ Doubly-linked list based deque implementation. """
```

```
def __init__(self):
    self._size = 0
    self._front = None
    self._rear = None
```

```
def addRear(self, newItem):
    """ Adds the newItem to the rear of the Deque.
        Precondition: none """
```

```
temp = Node2Way(newItem)
temp.setPrevious(self._rear)
if self._size == 0:
    self._front = temp
else:
    self._rear.setNext(temp)
self._rear = temp
self._size += 1
```

```
class Node:
    def __init__(self, initdata):
        self.data = initdata
        self.next = None

    def getData(self):
        return self.data

    def getNext(self):
        return self.next

    def setData(self, newdata):
        self.data = newdata

    def setNext(self, newnext):
        self.next = newnext
```

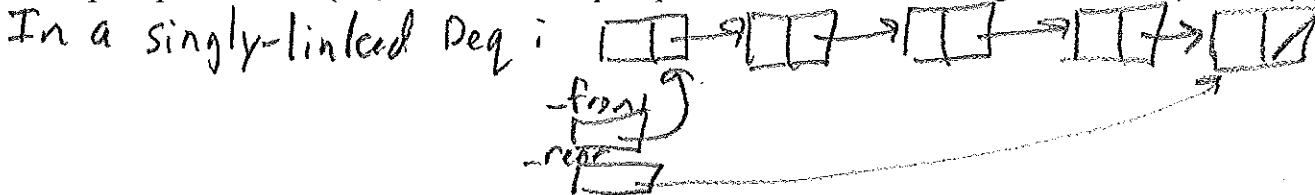
```
from node import Node

class Node2Way(Node):
    def __init__(self, initdata):
        Node.__init__(self, initdata)
        self.previous = None

    def getPrevious(self):
        return self.previous

    def setPrevious(self, newprevious):
        self.previous = newprevious
```

c) (5 points) Why would using singly-linked nodes (i.e., only `Node` objects with `data` and `next`) to implement the Deque lead to poor performance (i.e., cause some Deque operations to have worse big-oh notations)? Justify your answer.



When `removeRear` is called, resetting the `_rear` to point to the Node to the "left" of the one being removed is $O(n)$